SAVONIA

THESIS - **BACHELOR'S DEGREE PROGRAMME**
TECHNOLOGY, COMMUNICATION AND TRANSPORT

IoT service based on BLE technology

Author/s: Bainak Dmitrii

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Information Technology			
Author(s) Bainak Dmitrii			
Title of Thesis IoT Service Based on BLE Technology			
Date	01 May 2018	Pages/Appendices	39
Supervisor(s) Mr Arto Toppinen, Principal Lecturer			
Client Organisation /Partners			

Abstract

The purpose of this thesis was to create an IoT service capable of transmitting the temperature and the carbon dioxide measurements over BLE to a custom Android application. According to the plan, the Arduino platform, the BLE shield, the temperature and the gas sensor had to be combined as a single unit, which would measure the data and send it to the application. The app was aimed to display the measurements on the graphs and to exchange the data with the Savonia Measurements server. Moreover, it was planned to get familiar with the BLE protocol, the working priciples of the utilized sensors and some of the best practices of Android application development.

Initially, the Arduino Uno board and Arduino programming were studied to comprehend the Arduino platform and its features. Next, the specifications and the basic concepts of the BLE protocol were explored to understand how the transmission of data via BLE occurs. Afterwards, a research about the BLE shield and its features was conducted in order to expand the capabilities of the Arduino Uno board by mounting the shield on top of the board. Then the sensors were studied and attached to the board. Subsequently, the board was programmed so that it could measure the temperature and the carbon dioxide through the sensors and exchange the measurements with the help of the BLE shield. Finally, the Android application was created according to some of the best practices of modern mobile software devolepment, providing the properly functioning graphs and a stable connection with the Savonia Measurement server.

As a result of this thesis, a fully functioning IoT service based on BLE technology was implemented. Additionally, knowledge about the Arduino platform, the BLE shield, the aforementioned sensors, and the modern Android application development was acquired.

Keywords
Android, BLE, IoT, Arduino, MQ-135, temperature, gas, sensor, carbon dioxide, SaMi, MVVM, LiveData, ViewModel, Retrofit, GraphView, BLE shield

CONTENTS

1 INTRODUCTION

Nowadays, the IoT, which stands for the Internet of Things, is rapidly expanding on the global scale, encompassing various fields from smart home appliances and personal health monitoring systems to real-time tracking of environmental changes and complex manufacturing processes. The IoT represents a network of electronic devices, software components and services that can exchange data with the Internet or/and with each other. Wi-Fi and Bluetooth Low Energy (BLE) are the most widely used technologies for communication of such devices. Since smart phones play a significant role in our lives and are handy to utilize for different purposes, IoT devices are quite often paired with them. Smart phones allow to control the features of the devices and modify their settings, such as data exchange rate, power consumption, availability etc. The IoT keeps growing and new applications are being found every day, especially in automation. “The number of connected IoT devices will jump 12% on average annually. From nearly 27 billion in 2017 to 125 billion in 2030” – according to the analysis from IHS Markit, the leading source of information analytics in critical areas of technology. (IHS Markit 2017). Therefore, conducting practical research in this technological field appears to be immensely promising.

This thesis describes an implementation of an IoT service based on BLE technology. It will contain an Android application, an Arduino Uno microcontroller, a BLE shield v2.1 produced by RedBearLab, and a temperature sensor v1.2 and an MQ-135 gas sensor. The main goals of this project are to become familiar with Bluetooth Low Energy (BLE) protocol, to create an Android application by following some of the best practices of the modern mobile software development and to get a superficial insight into Arduino programming, circuit analysis and working principles of the aforementioned sensors. In the end, the service should be able to measure the temperature (C^0) and the carbon dioxide level (ppm) with help of the sensors, attached to the Arduino Uno board; the BLE shield should be mounted on the board in order to allow the transmission of the acquired measurements through its BLE module to other devices. The Android application should be able to subscribe to notifications with the measurements from the BLE shield and output the data on graphs. The next step is the establishment of the connection with the SaMi cloud service and the exchange of the data with it. The SaMi is a Savonia measurements website that supports interfaces allowing to save and retrieve different types of measurements. The application should also be able to retrieve the data stored in the SaMi cloud and to show it on the graphs.

The final prototype could be used as a temperature and air quality monitoring system for warehouses, offices and other facilities. For instance, the monitoring and processing of temperature and CO_2 changes throughout the day might be suitable for automated tweaking of air conditioners, oxygen concentrators and central heating systems depending on the current state of the values being measured. This would lead to a comfortable, productive environment, reduction of electricity costs and minimized deterioration of the equipment in the long term.

2 OVERVIEW OF COMPONENTS

In order to comprehend the functionality of the project, the reader should get familiar with specifications of the hardware and the software behind the scenes. This chapter is dedicated to the description of the Arduino Uno, the BLE shield, the BLE technology, characteristics of the sensors, SaMi cloud service, and an overview of application fundamentals in Android.

2.1 Arduino Uno

Since a detailed hardware analysis is not a part of this work, this subchapter provides a basic overview of the Arduino Uno. Additional information is available at Arduino website in case the reader is interested in indepth description.

Arduino Uno (Figure 1) is a microcontroller board based on the ATmega328P microprocessor. The microprocessor has 32 KB of flash memory, SRAM of two KB and one KB of EEPROM. The board also includes 14 digital input/output pins (six of which provide pulse width modulation output), six analog inputs, a 16 MHz quartz crystal, a USB port, a power jack input, an ICSP header, 13 built-in LEDs and a reset button. Its operating voltage equals 5V and the recommended input voltage varies from 7V to 12V; DC per I/O pin is 20mA. This board is the most popular, robust, beginner-friendly and well-documented amongst other versions of Arduino, therefore it perfectly suits the needs of this project. (Official Arduino web store 2018-05-02.)

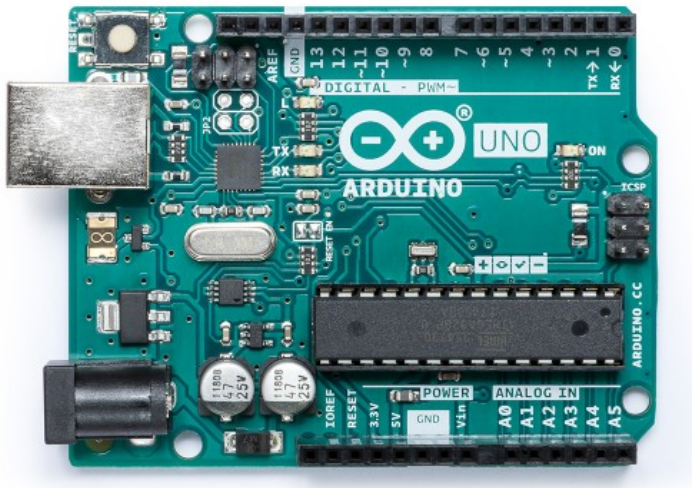

FIGURE 1. Arduino Uno appearance (Official Arduino web store 2018-05-02).

2.2 BLE shield

BLE shield stands for Bluetooth Low Energy shield. This piece of hardware is manufactured by RedBearLab for Arduino boards and compatibles. It extends capabilities of a board by allowing it to get connected with a central BLE device, for example, a smartphone with Bluetooth 4.0 or higher hardware support. It allows the board to send data to the connected device and use it as a getaway to the Internet. Moreover, the Arduino pins can be controlled with a custom application running on a smartphone. (Web store of Seeed Technology 2018-05-02.)

The BLE shield V2.1 (Figure 2) was selected for the current project because it is compatible with the previously described Arduino Uno board and capable of transmitting the measurements from the board to the Android application. The shield operates under either 3.3V or 5V. Its main component is a highly integrated single-chip Bluetooth® low energy connectivity IC, called nRF8001. This chip is specially designed for Bluetooth low energy applications targeted to work in Peripheral (Slave) role. Also, it has the lowest power consumption amongst its analogs. (Web store of Seeed Technology 2018-05-02.)

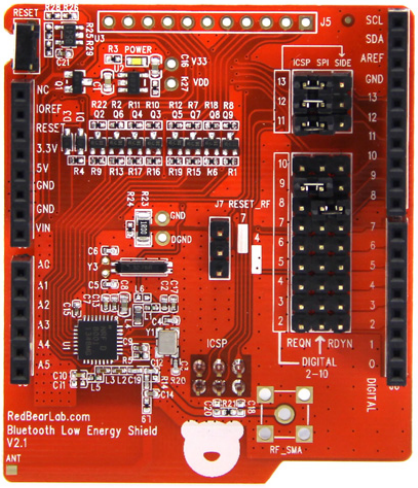

FIGURE 2. The appearance of Bluetooth Low Energy shield V2.1 (Official RedBearLab store 2018).

Communication between the shield and the board is implemented through the Application Controller Interface (ACI). The data exchange through the ACI is based on the Serial Peripheral Interface bus (SPI), which is a synchronous serial interface. The whole physical interface of the ACI consists of five pins: MISO, MOSI, SCK, REQN, and RDYN (Refer to Table 1 for the clarification). The SPI has a master-slave architecture with a single master. The shield acts as a slave with only one stipulation: since the shield may receive data at any time, be occupied with a connection event or data processing, all the transactions are controlled through two active low hand-shake signals via RDYN and

REQN pins. These pins can be flexibly selected from pin 2 to pin 10 of the board. When the board requests data from the shield, it puts the REQN to low, until RDYN is set to low by the shield. Afterwards, the master generates the clock for reading data. When the data is read by the master, it puts REQN to high. Next, the shield replies by setting the RDYN to high. The exchange of data occurs through MOSI and MISO pins and the generated clock is provided through SCK pin. In case the shield has to send the data to the master, it sets RDYN to low, whereas the master reacts to this change by setting REQN to low and generating the clock to read out the data. When master finishes reading the data, it puts REQN to high and the shield follows by setting RDYN to high, too. (Web store of Seeed Technology 2018-05-02.)

Commented [A1]: REQN is 9th digital pin of arduino RDYN is the 8th one

TABLE 1. Description of the ACI pins (Web store of Seed Technology 2018-05-02).

Signal	Arduino	nRF8001	Description
MISO	Input	Output	SPI: Master In Slave Out
MOSI	Output	Input	SPI: Master Out Slave In
SCK	Output	Input	SPI: Serial Data Clock
REQN	Output	Input	Handshake signal from the board to the shield
RDYN	Input	Output	Handshake signal from the shield to the board

2.3 BLE technology

The reader should gain insight into BLE protocol terminology to understand how the exchange of the data occurs and what requirements should be met to process the data. There is no way to provide a brief introduction to the full BLE protocol stack, therefore this subchapter describes only essential parts of the protocol, which will be discussed or used throughout the implementation of the project.

Bluetooth low energy wireless technology was announced in 2010 by the Bluetooth Special Interest Group (SIG) as a feature of the Bluetooth Core Specification Version 4.0. It was aimed to advance markets of devices requiring wireless connectivity that would keep power consumption and costs at low rates. It transfers the data packets with a rate of 1 Mbps through the license-free 2.4GHz Industrial Scientific Medical (ISM) band. (Bluetooth SIG 2016-12-06, 323.) Moreover, full AES-128 encryption results in strong security and authentication of data packets. (Bluetooth SIG 2016-12-06, 2298).

Since Local Area Networks and other applications based on IEEE 802.11 specification use the same ISM band, Bluetooth uses Adaptive Frequency Hopping (AFH) to overcome the interference of other devices and increase the amount of successfully transferred packets. Frequency Hopping (FH) is a technique of assigning unique hop sequences of channels when devices create a link. When the link is being formed, the devices are synchronized to change the channels repeatedly according to the created pattern of channels. AFH is an advanced type of FH, which avoids channels that are being used by other devices simultaneously operating in the same ISM band and adapts the hop sequence appropriately. (Bluetooth SIG 2016-12-06, 258, 536-539.) Moreover, the popular Cyclic Redundancy

Check (CRC) is implemented on received packages to improve the overall robustness against interference. (Bluetooth SIG 2016-12-06, 192-194).

2.3.1 Generic Access Profile

A BLE device can transfer information to other devices either through broadcasting or through connecting. The full specification of these mechanisms is described by Generic Access Profile (GAP). The description of the project is focused on the second option, since the BLE shield can exchange data only with connected devices. For connecting devices GAP defines Central and Peripheral roles. Central devices are typically phones, tablets or PCs with higher CPU's processing powers, whereas peripheral devices are represented by diverse low power devices and sensors, which connect to the central device. Peripheral devices usually use GAP to advertise data payload to let observing central devices in range discover them. When a peripheral device is connected to a central device, it stops advertising GAP data payloads. Notice that only one central device can be connected to a peripheral device. (Bluetooth SIG 2016-12-06, 1982-1986.).

2.3.2 Attribute Protocol

Bluetooth Low Energy has two core specifications: Attribute Protocol (ATT) and Generic Attribute Protocol (GATT). ATT is a low-level profile defining how to transfer units of data (attributes). ATT also defines the communication between two devices that can play server and client roles independent of the peripheral or central roles. Thus, during the data exchange, they can swap client and server positions. Services, characteristics, and related data are stored in a lookup table of a server. The client can make requests to read, write, notify or indicate attributes. These requests trigger response messages from the server. Moreover, the server can send two types of messages with attributes: unconfirmed notifications and indications that require the client to send a confirmation. (Bluetooth SIG 2016-12-06, 2173-2178.)

2.3.3 Generic Attribute Protocol

GATT is a high-level layer, which resides on top of the ATT. It defines the hierarchical data structure (Figure 3) that can be viewed as a meta-layer of the ATT that the server holds. Profiles, Services, Characteristics, and Descriptors are nested objects of the GATT. Profiles are virtual groupings of services that could be pre-compiled either by Bluetooth SIG or by manufacturers of BLE devices. Profiles define use cases for different types of devices in terms of their functionality and things they can do. A service is a collection of logically separated entities of data called characteristics. Usually, services represent features of devices. Services, characteristics and descriptors are all types of attributes that are distinguished by their Universal Unique Identifiers (UUIDs). Official Bluetooth adopted attributes have 16-bit UUIDs, while the custom ones get 128-bit UUIDs assigned. (Bluetooth SIG 2016-12-06, 2225-2230.)

Each entry in the lookup table of the GATT server consists of:

- Handle – the address of the Attribute, which is accessed via ATT

- UUID - Universally Unique Identifier, describing the attribute type
- Value – an array of bytes that is interpreted depending on the UUID
- Permissions – if and how the client can access the current attribute (Bluetooth SIG 2016-12-06, 2225-2228.)

Characteristics are the main data points of the GATT, since they can contain configuration or current state of a device or its components, for instance, measurements and parameters of the device's sensors. A characteristic contains a type, a value, descriptors and optional properties and permissions. UUID of a characteristic is used to identify its type, as in case of services. A value contains a unit of data. Properties define operations that can be performed on a characteristic by a paired device, such as READ, WRITE, NOTIFY or INDICATE. Reading a value transfers it from an attribute table of a server to a client over Bluetooth. Writing the value modifies its state; it is useful when the client needs to change, for example, configuration parameters of a sensor. Notifications can be sent to the client periodically or whenever the characteristic's value changes. In case an indication is sent by the server, it waits for a delivery acknowledgement from the connected client before sending the next indication. Both NOTIFY and INDICATE properties require the client to enable them firstly through Client Configuration Characteristic Descriptor (CCCD). Permissions are used for security purposes. They define whether the client can read or write characteristics, describe a level of encryption and authorization. Characteristic descriptors provide the client with additional information about the characteristic and its value. They are also identified by UUIDs and can be vendor-specific or adapted by Bluetooth SIG. (Bluetooth SIG 2016-12-06, 2235-2239.)

The only descriptor, which is essential for the project is the aforementioned CCCD. It is the most important and widely used descriptor since it regulates server-initiated updates of its parent characteristic. It is comprised of two-bit bitfield that can be set and cleared by the client at any time. The first bit is responsible for notifications and the second one corresponds to indications. Thus, whenever a client decides to enable notification of a certain server's characteristic, it simply sends Write Request ATT packet to set the first bit to 1 or uses 0 to disable the notification. (Bluetooth SIG 2016, 2238-2239.)

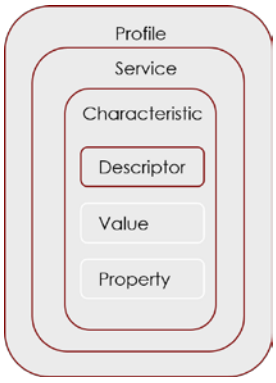

FIGURE 3. Hierarchical data structure of the GATT (self-made)

2.4 Overview of the sensors

2.4.1 Gas sensor

MQ-135 is a gas sensor suitable for detection of ammonia, sulfur, benzene, carbon dioxide, smoke, alcohol and other harmful gases. It consists of a micro AL_2O_3 ceramic tube, a gas sensing layer of Tin Dioxide (SnO_2), a heater coil and a measuring electrode mounted into a net, which is made out of plastic and stainless steel. The heater creates working conditions for the sensing layer allowing the sensor to change its analog output depending on the concentration of gases around it. The more concentration of gases the sensor is exposed to, the more output voltage it provides. Its detecting concentration scope varies from 10 to 1000 particles per million (ppm). The standard detecting conditions of the sensor are 24 hours of preheat time, temperature around 20°C, 65% of humidity, 5V of the input voltage and 5V of the output voltage. The sensor has a low cost (around 6 dollars), fast response and high sensitivity, stable and long life making it a great option for various air quality control applications. The sensor is mounted on the module (Figure 4) with an LM358 dual operational amplifier, a power LED, a LED for digital output, four pins and a potentiometer for adjusting the sensitivity of the digital LED. The module significantly simplifies the usage of the sensor and allows it to be used out of the box. (Waveshare wiki 2015-07-25.)

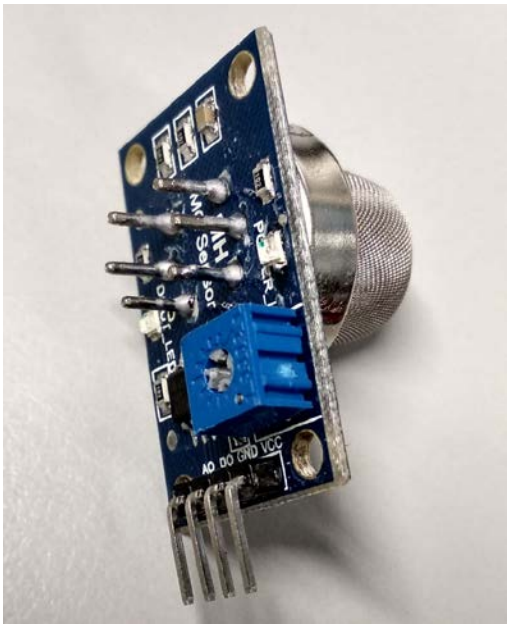

FIGURE 4. Photograph of the MQ-135 gas sensor module (self-made)

2.4.2 Temperature sensor

The temperature sensor V1.2 (Figure 5) manufactured by Grove uses an NCP18WF104F03RC (NTC) thermistor to measure the ambient temperature. Its working principle is based on the resistance of the thermistor. The more the ambient temperature rises, the less the resistance of the thermistor becomes. Its operating voltage ranges from 3.3V to 5V, the zero power resistance is 100000 Ohm and the operating temperature varies from -40 to +125 C° with an accuracy of 1.5 C°. (Seed Wiki 2018-05-10.)

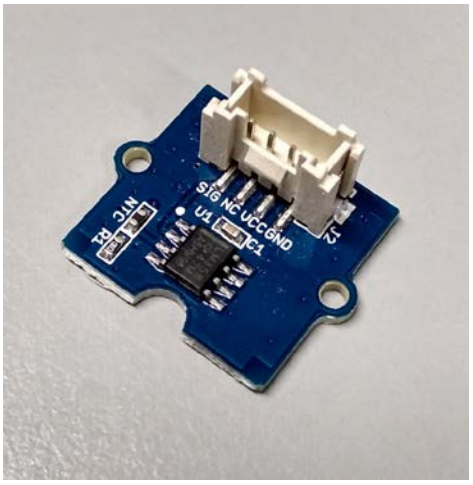

FIGURE 5. Photograph of the temperature sensor V1.2 (self-made)

2.5 SaMi cloud service

The SaMi service has a JSON interface for the data exchange with its clients. It allows them to send POST requests with two parameters: a key for unique identification of a user and a JSON array with measurements, which should be built according to the specific measurements data models. Alternatively, it is possible to retrieve the saved data from the cloud by sending a GET request with the use of the same key and multiple search parameters. The retrieved data can be converted from the JSON array to a Java object in accordance with the same data model. (Savonia Measurements).

2.6 Android OS

Android applications can be written in Java, Kotlin or C++ programming languages. All the code files, resources, and other data are compiled by the Android SDK tools into an APK file, which could be installed on Android-powered devices. The application is written in Java programming language using Android Studio IDE. Hence, the most part of the further details will be dedicated only to Java-based application development in Android Studio. (Vasconcelos 2016, 2-5.)

Basically, an Android operating system consists of native C/C++ libraries and Java API frameworks along with Dalvik Virtual Machine (VM) or Android runtime (ART) on top of the Linux kernel. Each app acts as a separate Linux user with its unique Linux user ID, its own Linux process, and permissions. Furthermore, each process runs in a separate virtual machine in isolation from other processes. (Vasconcelos 2016, 2-6.)

2.6.1 App modules

Each Android studio project contains source code files, assets, test code files and build-related files. A project could be divided into distinct collections of source files and build settings called Modules. Types of modules include **Android app module**, **Library module**, and **Google Cloud Module**. (Google Developers 2018-05-13, Projects overview).

Each Android app module is separated into three categories: **manifests**, **java**, and **res**. Within **manifests**, there is '**AndroidManifest.xml**' file, where essential information about the app is specified. It is used by Google Play, Android OS, and Android build tools. Permissions, the name of the application package, hardware and software features required by the app and the app's components are declared in this file. **Java** contains packages with java source code files and files with JUnit test code. **Res** is used for storing additional resources, such as XML layout files, UI styles, colors and strings and bitmap images. (Google Developers 2018-05-13, Projects overview.)

Android Library module is a container filled with reusable parts of code resembling Android app modules. These Library modules are compiled into archive files, which are easy to include in several app modules or import from other projects. Android studio allows creating two types of libraries: Java Libraries and Android Libraries. An Android library may contain source code files, Android manifest and resource files. This module provides a possibility to build different applications using various combinations of same reusable components. Java Libraries provide access only to Java source files. (Google Developers 2018-05-13, Projects overview.)

Google Cloud Module is used for managing Google Cloud backend code, but it is irrelevant for the project. (Google Developers 2018-05-13, Projects overview).

2.6.2 Application components

There are four essential app components: **Activities**, **Services**, **Broadcast receivers** and **Content providers**. The latter two are irrelevant for the project, therefore their description will be intentionally omitted. Each component has its unique lifecycle and purpose. Lifecycles of the components define their states during the execution of an app. These states depend on manipulations of the system and actions of a user. In order to use a component in the app, its information should be registered in the manifest file of the application. (Google Developers 2018-05-13, Application Fundamentals.)

An Activity is the main point for interaction with the user, which represents a single window with a user interface. It keeps the foreground focus on a current task, outputs its progress, reacts to the user's manipulations, invokes other activities, and communicates with other components on demand. All activities must be implemented as subclasses of the "Activity" class. Generally, an app contains a **main activity**, which, as the name implies, plays the major role in the application workflow and acts as the entry point for the application. At the same time, depending on the implementation, it is possible to invoke other activities of the app from separate applications without calling the main activity. (Google Developers 2018-05-13, Application Fundamentals.)

A Service is a component without UI that keeps the app running in the background. It is utilized for performance-demanding and long-running operations or execution of tasks for remote processes. For instance, a service could be uploading, archiving or syncing some data. Services are divided into three categories: **Foreground**, **Background**, and **Bound**. All services must be implemented as subclasses of the "Service" class. (Google Developers 2018-05-13, Application Fundamentals.)

Foreground services execute processes in such a way that the user is aware of them. A running foreground service must display a visible to the user **Notification,** even when the user does not explicitly interact with the application, to which the foreground service belongs. Foreground services are typically used to display a progress dialogue of a downloading process or a music playback. Such services can not be destroyed by the system, allowing them to contribute to the smooth user experience. (Google Developers 2018-05-13, Application Fundamentals.)

Background services perform operations that are not accompanied by any kind of visual notification. The most obvious example is a pedometer counting steps in the backround during the whole day. Nevertheless, background services can be destroyed by the system to release required resources if there is insufficient amount of RAM or processing power. (Google Developers 2018-05-13, Application Fundamentals.)

A Bound service acts as a server in a server-client interface. It performs operations as long as there is at least one application component bound to it via "bindService()" call. Only bound components can interact with the bound service. If all the bound components unbind from the service, it gets destroyed. Also, it is possible to create a **started bound service** by implementing two callback

methods in it: “onStartCommand()“ and “onBind()”. This allows to start the indefinitely running service, to which other components can bind. Such service should be stopped by explicitly invoking either “stopSelf()” or “stopService()” methods. (Google Developers 2018-05-13, Services overview.)

2.6.3 Gradle and API levels

Android studio IDE uses Gradle build system to customize and configure builds and create various APK files from one project by combining specific modules. Moreover, Gradle simplifies reusability of files from different sourcesets. It is helpful for building different versions of one app from a single project with limited or full functionality. Build dependencies on modules, local and remote binaries can be configured in the ‘**build.gradle**’ file of a project. (Google Developers 2018-05-13, Application Fundamentals.)

Each version of Android provides only one API level, which consists of permissions, core packages, classes, XML elements, attributes etc. New versions of APIs remain forward compatible with the previous ones, which means that applications written for older APIs could be installed and utilized on devices with newer APIs. Backward compatibility could be missing in case an Android application is compiled against a new API level with the use of new additional frameworks, which are missing from the previous API versions. Consequently, during application development a programmer should specify minimum, maximum and target API levels in the application’s manifest file to filter the app from devices that do not correspond Android version requirements. (Google Developers 2018-05-13, Application Fundamentals.)

3 ARDUINO, BLE SHIELD AND SENSORS

3.1 Mounting the shield

To establish the communication between the Arduino Uno and the BLE shield through the Application Controller Interface (ACI), there were REQN and RDYN jumpers set on the shield. Thus, the ninth digital pin of the Slave was connected to the REQN pin of the Master, while the eighth digital pin of the board was selected for the communication with the RDYN pin of the BLE shield (Figure 6). Afterwards, the shield was physically mounted on top of the board.

FIGURE 6. REQN and RDYN pins
(Official RedBearLab web store 2018-05-15.)

3.2 Connecting and programming the sensors

3.2.1 Temperature sensor

The temperature sensor has four pins: SIG, NC, VCC, and GND. The SIG pin of the sensor was connected to the fifth analog pin (A5) of the Arduino board. Next, the sensor's NC pin was attached to the NC pin of the microcontroller. The 5V pin of the Arduino Uno provides the sensor's VCC pin with the power. Finally, the sensor's GND pin is grounded to the GND pin of the board.

The measurements from the sensor are delivered to the A5 pin of the board, where the following code (Figure 7) computes the temperature according to the datasheet of the sensor:

```
// temperature reading
int a = analogRead(pinTempSensor);

float R = 1023.0/a-1.0;
R = R0*R;

double temperature = 1.0/(log(R/R0)/B+1/298.15)-273.15; // convert to temperature via datasheet
Serial.print("Temperature sensor reading: ");
Serial.println(temperature);
```

FIGURE 7. Code calculating the temperature (Seeed wiki 2018-05-10.)

3.2.2 Gas sensor

The gas sensor is also connected to the 5V and the GND pins of the board. Its digital pin is wired to the third digital pin of the Arduino Uno. The analog pin of the sensor is connected to the 21.4 KOhm load resitor (R_L) and the A0 port of the microcontroller. The load resistor is also connected to the ground. Hence, it acts as a pull-down resistor.

To measure the ppm of the CO_2, the Arduino sketch uses the “MQ135” library. According to the datasheet, it is possible to calculate the R_0 by measuring the resistance output of the sensor (R_s) at a certain concentraition of the carbon dioxide. Nowadays, the known concentration of the CO_2 in the atmosphere equals 410.31 ppm. Hence, preheating the sensor for 24 hours allowed to calibrate it properly and derive the R_0 with help of the “MQ135” library. Afterwards, the R_0 value within the header file of the library was changed to the derived value. Consequently, the library became capable of calculating the actual concentration (ppm) of the carbon dioxide.

3.3 Sending the measurements via BLE

The Arduino board makes use of the “RBL_nRF8001” and “BLE SDK for Arduino” libraries to send the measurements via the BLE shield. They were added to the IDE through the Arduino Library Manager and then included into the existing sketch. The API of the first library significantly simplified the programming routine, because it was required to make only a few calls to the API in order to send the data via BLE. This is the list of the APIs that were utilized in the sketch:

- “ble_begin()” - prepares the BLE stack and enables the broadcasting of the advertising packets;
- “ble_set_name()” - changes the broadcasted name of the BLE shield;
- “ble_write_bytes()” - writes an array of bytes to the Master. This API was used to write the actual measurements that are sent by the shield within one of its charateristics;
- “ble_do_events” - notifies the Master that it should processes its events and send the data that have been previously written to it. (REDBEARLAB 2018-05-15, ‘README.md’ file of ‘nRF8001’ GitHub repository).

4 ANDROID APPLICATION

The GitHub repository of the Android application can be accessed via this link: https://github.com/bajnax/Thesis

4.1 Libraries

First of all, the development of the application began with the selection of suitable libraries. Since the Android OS has a built-in API for the BLE, the main emphasis was made on other objectives of the project, namely the output of the temperature and gas measurements on the graphs and the exchange of the data with the SaMi cloud service. Android **Graph View** plotting library (v4.2.2) was selected for the former objective. Next, **Retrofit2** (v2.4.0) library and its **GSON** converter were chosen to implement an HTTP client and meet the demands of the latter one. Furthermore, after a thorough consideration of the architecture and the design of the app, "**Android Design Support Library**" (27.1.1) and a collection of libraries called "**Android Architecture Components**" were included into the project.

4.1.1 Graph View library

Graph View is an open source project targeted for plotting various diagrams. It is characterized by rich functionality, deep and flexible customization, and simplicity of integration. In particular, Graph View provides the XML integration, a combination of multiple and mixed types of series, tap listeners, a variety of styles, scrolling, zooming, secondary scales, and the formatting of the viewport and labels. Nonetheless, it lets swiftly create different types of graphs omitting in-depth studying of the library. In this project the **Point Graph** was selected for the data output as the most appropriate and convenient plotting type for the visualization of the gas and temperature measurements. Besides all the customizable layot-related features, this library offers a realtime/live chart drawing capability by means of appending new data points to the plotted data series or by resetting the data series completely as soon as new datapoints are passed to the corresponding methods. (Gehring 2018-05-16, "Summary&Features" tab of the GraphView website).

Realtime updates were utilized in the thesis project depending on the arrival of the BLE notifications with the measurements. Thereafter a custom label formatter was implemented to accompany synchronously each realtime update with an addition of a time label on X-axes. Additionally, the graphs of the application make use of the tap listeners on the data points so that the listeners get invoked when the user taps on the data points. The tap listeners show toasts (fading dialogues) with the information corresponding to the tapped point (Figure 8).

The graphs can be added to layout files in the same way as any other views. To include the library into the application, the following lines should be added to the dependencies list of the "**build.gradle**" file of the application module:

implementation 'com.jjoe64:graphview:4.2.2'

implementation 'com.android.support:design:27.1.1'

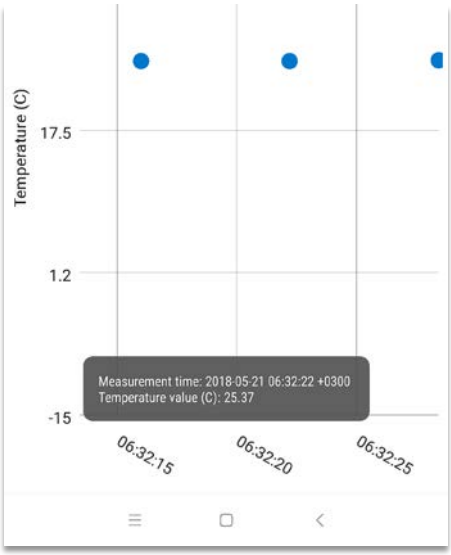

FIGURE 8. Screenshot of the app with the toast and the custom label formatter (self-made)

4.1.2 Retrofit2 library and its GSON converter

Retrofit is a type-safe HTTP client for Android and Java developed by Square. It is designed to simplify data excgange with REST-based web services by transforming HTTP APIs to Java interfaces. This library allows to configure various converters for data serialization and deserialization. Behind the scenes, Retrofit is built on top of OkHttp3 library, which supports it in handling HTTP requests. To meet the requirements of this project, GSON converter was used to convert Plain Old Java Objects (POJOs) to JSON objects in order to send POST requests. The conversion of JSONs to POJOs was utilized to process responses of GET request. (Square, Inc. 2013).

To make Retrofit and GSON function in a proper way, several simple steps were completed. Firstly, the libraries were added into the application module. Secondly, Java classes for sources were generated. Next, the construction of API's interface took place, followed by declaration of the Retrofit web client.

These are the libraries included into the dependencies list of the “build.gradle” file, which belongs to the app module:
implementation 'com.squareup.retrofit2:retrofit:2.4.0'
implementation 'com.squareup.retrofit2:converter-gson:2.4.0'

Java classes were written in accordance with object models that are provided in the instructions for the SaMi server. They are comprised of a “MeasurementsPackage”, a “MeasurementsModel”, and a “DataModel” classes. The “MeasurementsPackage” class is used to store a ‘key’ that uniquely identifies a user of the SaMi service and a list of objects of “MeasurementsModel” type. Each object of “MeasurementsModel” stores information about measurements and a list of objects of “DataModel” type. Whereas objects of “DataModel” type contain the sensor’s data. Since this project is dedicated to measurements coming from two sensors, each “MeasurementsModel” contains two objects of “DataModel” type. The first one holds temperature-related data and the second one keeps information about the carbon dioxide level. (Savonia Measurements 2018-05-12, Help page of the Savonia Measurements website.)

Whenever the application sends measurements to the SaMi server, it makes sure that the object models follow these requirements:

- “MeasurementsPackage” object contains a valid key and a list with objects of “MeasurementsModel” type
- Each “MeasurementsModel” object has a timestamp, which is created in compliance with ISO 8601 date and time format. Moreover, measurement name and measurements tag must be present in each object of “MeasurementsModel” type
- Objects of “DataModel” type must contain values of actual measurements and tags of the corresponding sensors. (Savonia Measurements 2018-05-12, Help page of the Savonia Measurements website.)

GET requests are easier to build, because the user has to mention only the key and the sensors tags that were used for saving data through POST requests.

For example, imagine a case when the app received five measurements of the temperature and five measurements of the CO_2 level and the user decides to send a POST request with this data to the SaMi server. In such a situation the first measurement of the temperature will be stored in one object of “DataModel” type and the first measurement of the CO_2 will be also saved in another object of of the same type. Next, these two objects are put in a list of ”DataModel” objects, which belongs to an object of “MeasurementsModel”. The second measurements of both the temperature and the CO_2 will be packed in the same manner. The packaging creates a list containing five objects of “MeasurementsModel” type, each holding two objects of ”DataModel” type. This list is inserted into an object of “MeasurementsPackage” type. Finally, the data can be sent to the SaMi. (Savonia Measurements 2018-05-12, Help page of the Savonia Measurements website).

4.2 Activity lifecycle

When a user pauses, resumes or closes an application, the app goes through various states in its lifecycle. There is a list of predefined callbacks provided by “Activity” class that are used to notify the activity about changes in its lifecycle (Figure 9). Within the methods of the corresponding callbacks, programmers adjust the behavior and handle the transitions of the app, release or request the resources depending on a current change of state. For instance, saving or restoring the user’s progress, pausing or resuming a video, or reconnecting to the internet are essential actions that should be implemented to avoid crashes of the app and improve the overall user experience and performance of the app. Additionally, the proper handling of the lifecycle callbacks is required due to the fact that activities frequently get destroyed and recreated on the configuration changes. These changes include a screen rotation, a switching to multi-window mode or a keyboard invocation. The recreation mechanism allows activities to adapt to new configurations by redrawing the user interface. Therefore, a combination of “onSaveInstanceState()” and ”onRestoreInstanceState()” methods, ViewModels, and a local storage was used during the development of the app for saving and restoring states of activities. (Google Developers 2018-05-19, Understand the Activity Lifecycle).

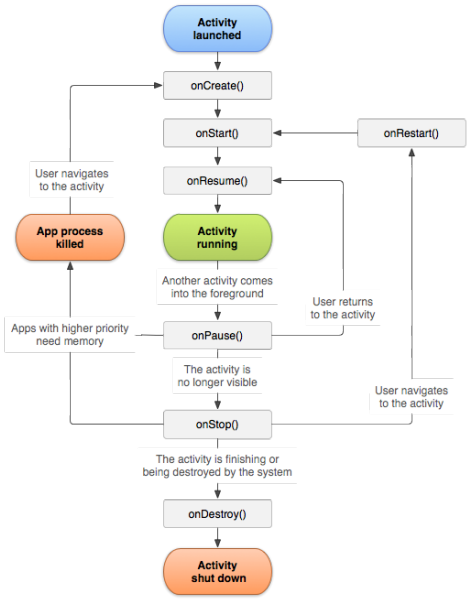

FIGURE 9. Basic representation of the activity lifecycle (Google Developers 2018-05-19, Activity lifecycle image.)

4.3 Fragment

The application of this project uses Fragments that are provided by the support library. A Fragment represents a behavior or a part of the user interface of an application. It must be running inside a hosting FragmentActivity. The Fragment has its own lifecycle (Figure 10), but it is closely tied to the lifecycle of the Activity. No fragments can be started in a stopped activity and they get destroyed as soon as their hosting activities are destroyed. At the same time, Fragments can be added to one Activity, removed from it or transitioned to another Activity during the execution of the app. Furthermore, the interface of a Fragment can be defined in a separate layout file. Interactions with the Fragments are implemented through FragmentManager, which can be obtained via call “getSupportFragmentManager()”. (Google Developers 2018-05-20, Fragments).

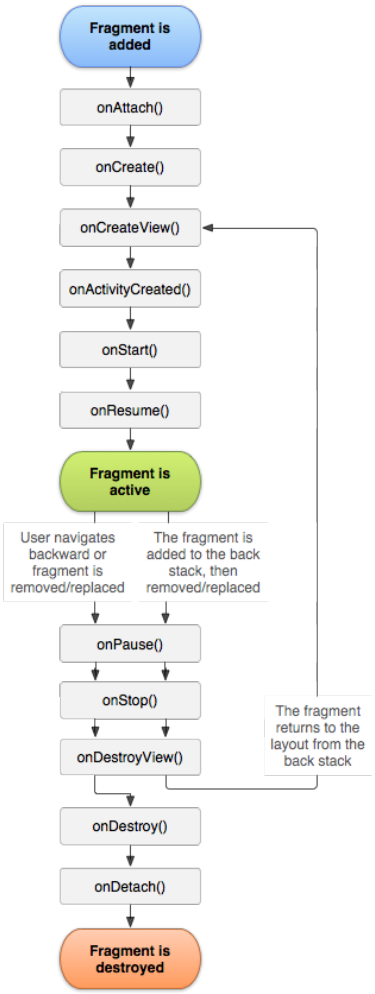

FIGURE 10. Basic representation of the fragment lifecycle (Google Developers 2018-05-20, Fragment lifecycle image.)

4.4 Architecture of the app

Model-View-ViewModel (MVVM) architectural pattern was implemented in the Android application of this project through the extensive use of "**Android Architecture Components**". MVVM pattern follows the 'separation of concerns' principle by dividing the application into separate independent parts. The Data Model represents an abstraction of the data source. It combines access to both local database and calls to the SaMi server in a single repository. The ViewModel exposes streams of events to the View, while the View notifies the ViewModel about the user's actions. Thus, a combination of the Data Model and the ViewModel allows to save and retrieve, send and receive data without holding a reference to the View (Figure 11).

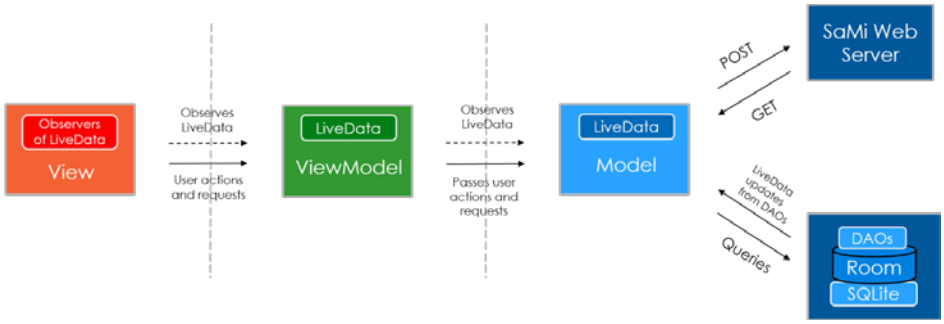

FIGURE 11. Schematic representation of the MVVM pattern (self-made)

4.4.1 LiveData

LiveData is a class, which is used for holding observable data in a lifecycle-aware manner. An "Observer" might have a paired "LifecycleOwner" represented by an activity, a service or a fragment. An object of "Observer" type can subscribe to updates of a LiveData object. The updates notify the observer whenever its "LifecycleOwner" has either "STARTED" or "RESUMED" state. Thus, the observer is able to update UI directly whenever there is a change of the app data. (Google Developers 2018-05-21, LiveData Overview.)

If the "LifecycleOwner" has another state, the "Observer" is considered inactive. Therefore, it does not get notified about changes of LiveData, and consequently stopping the application does not cause crashes. Also, in case the "LifecycleOwner" transitions to "DESTROYED" state, the "Observer" is unsubscribed from the updates of LiveData. This ensures that no memory leaks occur because of destroyed activities and fragments. (Google Developers 2018-05-21, LiveData Overview.)

LiveData has a subclass called MutableLiveData. MutableLiveData exposes two methods: "setValue()" and thread-safe "postValue()". These methods are used for forwarding the data to active Observers of LiveData. Additionally, MutableLiveData is a parent class for MediatarLiveData. The latter one can be used as an observer of other LiveData objects, compose them together and react

to their changes. MediatorLiveData allows to control which updates should be propagated to its observers. Hence, it is suitable for ViewModels as an intermediary between the DataModel and the View. (Google Developers 2018-05-21, LiveData Overview.)

LiveData is especially useful for restoring the data in the UI after configuration changes, because inactive components receive the latest update as soon as they return to their active state. (Google Developers 2018-05-21, LiveData Overview).

4.4.2 ViewModel

ViewModel is used to keep and control the View data in a lifecycle-aware way. Since UI controllers (activities and fragments) may be destroyed and recreated due to configuration changes, a resource cleanup implemented by the Android system, or because of the user's actions, they should implement the logic for saving and restoring the data. It allows to avoid unnecessary and repetitive reloading of the same data every time the UI controller is re-created. In some cases it is enough to use "onSaveInstanceState()" method to save the view hierarchy in a bundle and restore it afterwards inside the "onCreate()" callback, but the space of the bundle is limited. Hence, it is advised to store only relatively small amounts of serializable primitives and simple objects in the bundle of the method. At the same time, if the data handling is delegated only to the UI controllers, they swiftly get bloated and become hard to maintain, test and cleanup to avoid memory leaks. This happens due to frequent asynchronous calls, that UI controllers make to retrieve and save the data. In such cases, preserving the UI state data with help of other modules is an appropriate approach. (Google Developers 2018-05-24, ViewModel Overview.)

The design of the ViewModel allows the UI data survive configuration changes. During the initialization of the ViewModel, it gets scoped to the "Lifecycle" that is provided to "ViewModelProvider". The initialization is usually implemented in the "onCreate()" callback of the UI controller. If the "Lifecycle" is represented by an activity, the ViewModel is stored in memory until the activity is finished (Figure 12). Whereas a fragment's data is retained in the ViewModel until the fragment is detached. Note that ViewModel should not hold any references to the View. Instead, it is typically used in a pair with LiveData to expose streams of events for the Observers within UI controllers. (Google Developers 2018-05-24, ViewModel Overview.)

To sum up, whenever a UI controller is re-created, it receives the same instance of the ViewModel that was initialized during the initial creation of the UI controller. Thus, the ViewModel passes the same data to various instances of the same UI controllers. (Google Developers 2018-05-24, ViewModel Overview).

Moreover, ViewModels are suitable for communication between fragments. For example, two fregments inside one activity can share data and react to its changes by observing the same instance of the ViewModel. To achieve such functionality, the ViewModel must be initialized in the fragments through passing the context (via "getActivity()" method) of their hosting activity. As a

result, each fragment can work independently from the other within its own lifecycle and can be safely replaced or removed at any time. (Google Developers 2018-05-24, ViewModel reference).

Furthermore, ViewModel has a subclass called “AndroidViewModel”. It is used when a reference to the application “Context” is required inside the ViewModel. The context of the application can be retrieved through calling “getApplication()”. (Google Developers 2018-05-24, ViewModel Overview).

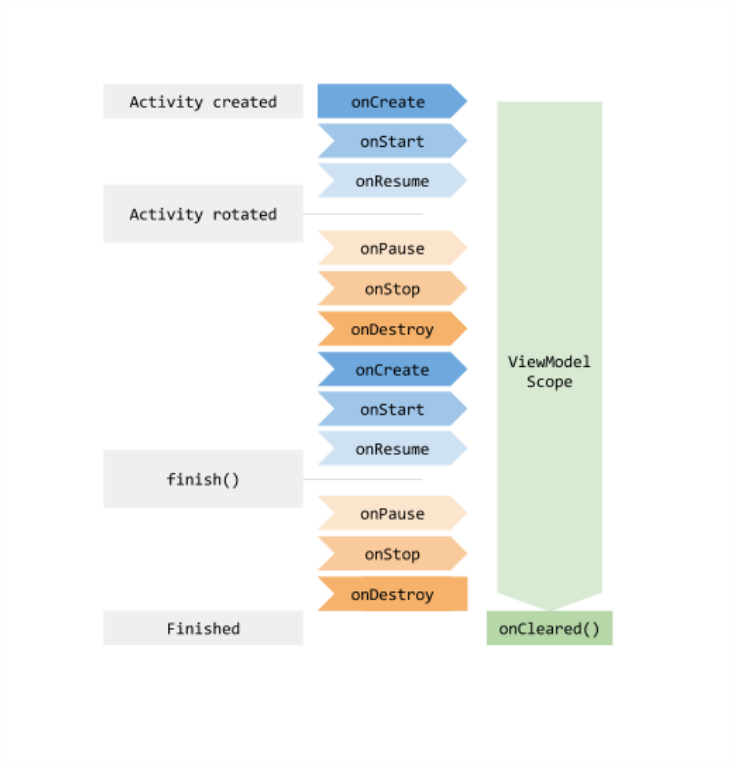

FIGURE 12. Lifetime of the ViewModel (Google Developers 2018-05-24, Lifetime of the ViewModel.)

4.4.3 The Room persistence

Room provides access to the full functionality of SQLite while acting as a smooth and simple abstraction layer on top of it. It allows to cache settings, search result and other pieces of relevant and structured data in a local database of a phone that runs an application with Room. Room consists of three main components: Database, Entity and DAO. (Google Developers 2018-05-27, Defining data using Room entities.)

Entities are classes that represent tables for the database. Each Entity class should be annotated with @Entity and at least one of its fields must be marked with @PrimaryKey annotation. Fields of

an entity act as columns of a corresponding table and can be also marked with various annotations. Moreover, entities may have relations, or logical connections, with each other (defined by foreign keys), since SQLite is a relational database. Each field of an entity should be either passed to a constructor, be public or have a public setter. (Google Developers 2018-05-27, Defining data using Room entities.)

DAO stands for Data Access Object. They can be included into a project as abstract classes or interfaces with @DAO annotations. A DAO class defines a layer used for interactions with a database and typically each entity has its own DAO class with specific methods. The methods can be marked with @Query, @Update, @Delete or @Insert annotations in order to handle corresponding queries. Additionally, it is a common practice to use LiveData within DAOs to build observable queries to databases. It is achieved by setting a return value of a query to LiveData type. Consequently, LiveData gets automatically updated by Room whenever there is a change in the database. DAOs are extremely useful because they simplify testing and allow to avoid lower level APIs for the communication with the database. Besides, all the quires are verified by Room during compilation, therefore runtime failures are easily avoided. (Google Developers 2018-05-27, Accessing data using Room DAOs).

The Database itself is represented by an abstract class that is annotated with @Database. This class must be a child of the “RoomDatabase” class. All the required DAO classes and entities should be defined in the database class. Furthermore, it is possible to mark the database with @TypeConverters annotation and define the type converters in it, allowing all DAOs and Entities to use them. The Room database is usually accessed through the Singleton pattern, its instance can be retrieved either via “Room.databaseBuilder” or “Room.inMemoryDatabaseBuilder”. (Google Developers 2018-05-27, Save data in a local database using Room).

4.5 Design

During the development of the application, the most part of the layout files was based on ConstrainedLayout provided as a support library. It was selected because of its rich functionality, ease of use and adaptive behavior for various screen sizes. Nevertheless, some of the Fragments are built with FrameLayout due to their simplicity. The consistent style prevails in each visible component of the app. Additionally, the simplicity of the interface and the efficiency of the application itself lead to the proper user experience.

4.5.1 ConstrainedLayout

A ConstraintLayout is a ViewGroup that is suitable for building flexible layouts. It allows to arrange and re-size widgets with regards to certain constraints. Some of the constraints are:

- Relative positioning – used to position a widget with relation to the position of another widget
- Margins – allow to avoid overlapping of the widgets by selecting a space between them

- Centering positioning and Bias – allow to center a widget and place it in the layout with a certain bias
- Visibility behavior – helps to define how a widget reacts to the disappearance of its neighbours
- Chains – define behavior for the groups of elements that are bi-directionally connected along either vertical or horizontal axes (Google Developers 2018-05-30, ConstraintLayout reference.)

4.6 Workflow and functionality of the application

4.6.1 Scanning activity

The starting point of the application is the activity ("LeScanActivity") that requests the user's permissions, scans for available BLE devices and outputs the found devices in a custom ListView.

Starting from the Android 6.0 (API level 23 or higher), it is required to request user's permissions during runtime of the application. This requirement was followed in the app to enable Bluetooth and location services of a phone running the application. The requests are shown as dialogues. If the user denies any request, the application does not work. Otherwise, granting permissions enables Bluetooth and location services.

Location services must be enabled due to a behavior of the "BluetoothLeScanner" class, which is used for BLE scanning. Therefore, "Google Location and Activity Recognition" library was utilized to build a dialogue window with the permission request (Figure 13). If the permission is granted, location services are enabled instantly.

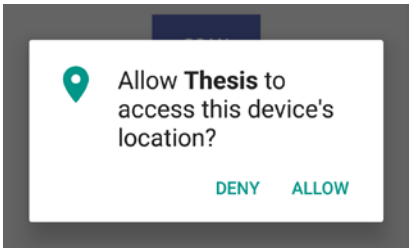

FIGURE 13. Screenshot of the dialogue with the permission request (self-made)

When the BluetoothLeScanner finds surrounding BLE devices, they are returned to its callback and shown in a custom ListView, where each list item displays a name of a device, its address and a "Connect" button (Figure 14). If the user clicks the button associated with the selected device, the device's information is passed to the next activity.

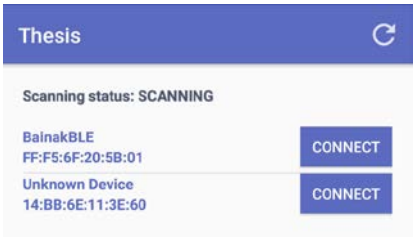

FIGURE 14. Screenshot of the "LeScanActivity"

4.6.2 Database

The database of the app stores two tables (“gas_table” and “temperature_table”), which are described in two classes annotated with @Entity. The first entity contains “id”, “GasValue” and “Date” columns along with their getters and setters. The second one holds “id”, “TemperatureValue” and “Date” columns with associated getters and setters. They define the data object models for persisting the corresponding measurements.

Each Entity has its own DAO interface. “GasDao” exposes four public methods:

1. “insert()” - saving one gas value in the “gas_table”
2. “deleteAll()” - allowing to delete all measurements of the gas from the table
3. “getAllGasValues()” - exposing a LiveData that holds a list of all the “Gas” values sorted by “id” column
4. “getAllGasValuesAsync” - returning a list of “Gas” values sorted by “id” column

“TemperatureDao” has a similar interface, which is used to manipulate “temperature_table” in the same way.

4.6.3 Retrofit web client

The functionality of the app’s Retrofit web client is based on three parts:

- Previously described object models of the SaMi
- “WebClient” class that is used to create a singleton instance of the Retrofit web client
- “SaMiClient” interface, which allows to access the HTTP API of the SaMi

“SaMiClient” interface contains two annotated methods for POST and GET request. “postMeasurements()” method accepts an argument of “MeasurementsPackage” type that is used as a POST request body. “getMeasurements()” method has a “key” parameter for accessing the user-specific data in the SaMi and several query parameters that are utilized to build unique GET requests. The response body of the second method is a list of “MeasurementsModel” objects. They are provided through responses of the successful GET requests.

4.6.4 Central repository

The “CentralRepository” class represents the core part of the app’s Data Model and acts as a single source of truth, because all the connection-related logic is kept in this file. Specifically, it holds the instance of the Retrofit web client and the instance of the database.

The repository contains several LiveData objects. One MediatorLiveData object (“mObservableGases”) holds a list of “Gas” objects, while another one (“mObservableTemperatures”) keeps a list of ”Temperature” objects. These MediatorLiveData objects observe changes in the database tables through their relative DAOs.

There are also several MutableLiveData objects holding:

- Response statuses of the HTTP requests
- Response body of the GET request
- Asynchronously retrieved list of “Gas” objects
- Asynchronously retrieved list of “Temperature” objects

The latter two are used for creating the bodies of the POST requests.
Furthermore, the repository contains nested “AsyncTask” classes and several public methods allowing to access them from other modules of the app. “ClearDbAsync” asynchronously clears the database. “GenerateTemperatureMeasurementAsyncTask” and “GenerateGasMeasurementAsyncTask” asynchronously retrieve the measurements from the database when the POST request body is being built.

4.6.5 ViewModel

The application has several ViewModel classes for various purposes. “SensorsDataViewModel” extends AndroidViewModel, because it holds a reference to the central repository retrieved through the application context. This class has MediatorLiveData objects, which observe the changes of the measurements through public methods of the central repository that, in turn, provide these objects with the Livedata updates. Additionally, new measurements can be inserted into the tables through these objects when the View invokes corresponding public methods of the ViewModel. At the same time, “SensorsDataViewModel” exposes public methods with LiveData that the View uses to observe the changes of the temperature and the gas values.

Another ViewModel is called “SaMiViewModel”. It is also connected to the central repository, but it carries out different tasks. Its main purpose is to pass query parameters of HTTP requests from the View to the repository and update the UI through LiveData by observing the repository’s LiveData holding the GET responses.

“SharedViewModel” provides a communication line between “LeConnectedDeviceActivity” and its fragments. A similar function is dedicated to “GetRequestViewModel” that is used by the “GetRequestActivity” and its nested fragments.

4.6.6 Activity with the connected device

When the user selects BLE shield in the list of the found BLE devices, “LeConnectedDeviceActivity” is started. This activity is responsible for displaying the data regarding the status of the BLE connection. It also displays the measurements on the graphs and allows to access activities for POST and GET requests through a popup menu in the toolbar. Moreover, the “Scroll to end” feature is accessible in the popup menu. It scrolls each realtime graph to the last added point, when the phone receives a new measurement.

The activity is divided into three Fragments: a fragment with the connection status, a fragment holding a graph with the temperature measurements, and a fragment that ouputs the gas values on the graph. The fragments are combined by a TabLayout and a ViewPager with disabled swipes. Clicking one of the tab icons on the top of the activity selects its corresponding fragment.

To observe the connection-related changes, the activity uses “BroadcastReciever” that receives GATT events from “BluetoothGattCallback” of the “BluetoothLowEnergyService”, which is a started bound service. These GATT events are further passed to the “ServicesFragment” that displays them on the UI. When the activity reaches the resumed state, it starts the service (if it is not running yet) and binds to it afterwards. Next, it passes the information about the selected BLE device to the service, where all the GATT manipulations reside. If the activity is stopped due to configuration changes, it unregisteres its receiver and unbinds from the service, but the service itself keeps running in the background. As soon as the activity is re-created, it registers the receiver and binds to the service once again. This ensures that simple rotation of the phone does not disrupt the connection. Alternatively, if the activity is stopped not because of configuration changes, it unregisters its receiver, unbinds from the service, and then stops it.

The first fragment called “ServicesFragment” (Figure 15) shows the connection status with the BLE shield. With help of this fragment the user can observe how the “BluetoothLowEnergyService” retrieves the GATT services and characteristics of the shield and then enables the characteristic notifications with the measurements.

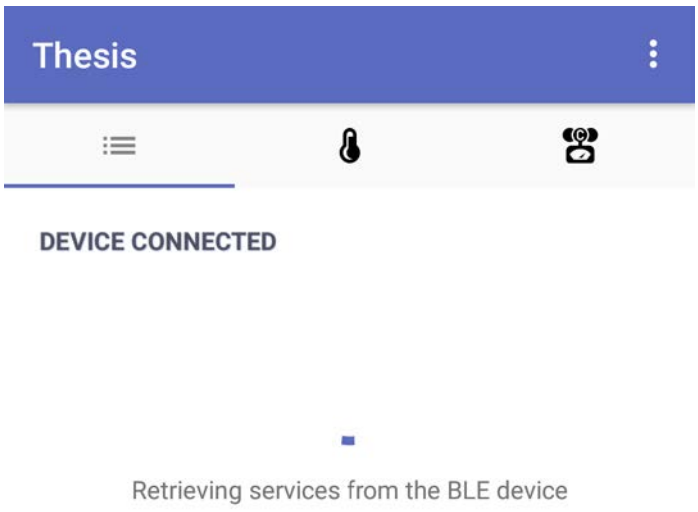

FIGURE 15. Screenshot of the “ServicesFragment” (self-made)

The “TemperatureFragment” (Figure 16) can be accessed through the middle tab of the activity. It holds the realtime point graph displaying the temperature measurements. The graph implements the tap listener on the data points. Moreover, the custom label formatter on the x-axes lets the user keep track of the date and time regarding the measurements acquisition. The updates with the measurements are observed through the “SensorsDataViewModel” that exposes LiveData to the View. Whenever a new measurement is saved in the database, it goes through the central repository to the “SensorsDataViewModel” and appears on the graph right away.

The last fragment is called the “GasFragment” (Figure 17). It works in the same way as the “TemperatureFragment”. The only difference is that it diplays the gas measurements and consequently holds the ‘ppm’ labels on the y-axes.

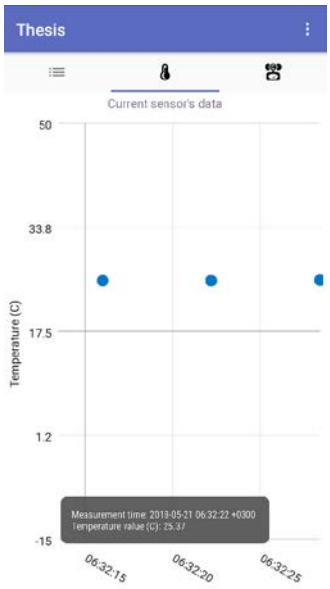

FIGURE 16. Screenshot of the ”TemperatureFragment” (self-made)

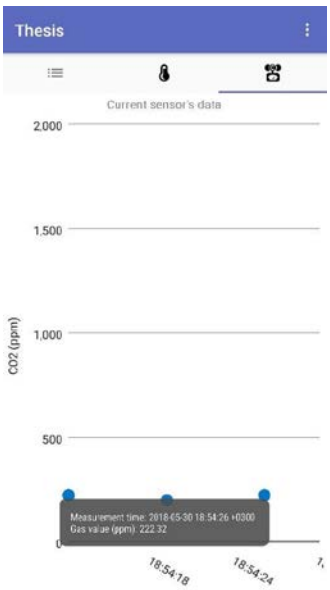

FIGURE 17. Screenshot of the ”GasFragment” (self-made)

Furthermore, both the “TemperatureFragment” and the “GasFragment” hold the same instance of the “SensorsDataViewModel” because they observe it with the context of their hosting activity. This leads to the improved performance of the app.

Additionally, since the data is persisted in the local database, every time the user opens the application, the previously acquired measurements are retrieved from the database and displayed on the graphs along with the new ones.

4.6.7 BLE service

The “BluetoothLowEnergyService” makes use of the Android BLE API. It uses “BluetoothGatt” to connect to the BLE shield and processes the communication events within the “BluetoothGattCallback”. Each GATT event invokes its corresponding overriden method of the “BluetoothGattCallback”. When the phone is connected to or disconnected from the BLE shield, the “onConnectionStateChanged()” method is invoked. Next, the method verifies whether the device is connected or disconnected. If it is disconnected, then the “BluetoothLowEnergyService” tries to reconnect to it. Otherwise, it starts the discovery of the services and their characteristics. As soon as the services and the characteristics are discovered, the “onServicesDiscovered()” method of the “BluetoothGattCallback” is invoked. Within this method the “BluetoothLowEnergyService” searches for the shield’s service that is responsible for providing characteristic with the measurements. When the service is found, the search for the measurements characteristic takes place. If this search is successful, then the “BluetoothGatt” writes the Client Characteristic Configuration Descriptor (CCCD) of the measurements characteristic to enable the characteristic’s notifications on the BLE shield. Hence, when the notification arrives, the “onCharacteristicChanged()” method of the “BluetoothGattCallback” is invoked. Finally, the received notifications are processed and saved into the database.

4.6.8 Activity for POST request

When the user selects “Offload data to SaMi cloud” option in the popup menu of the “LeConnectedDeviceActivity”, the “PostRequestActivity” (Figure 18) is started. This activity is used to build the data object model for the POST request. It offloads the whole database to the SaMi cloud and then clears the local database if the POST request is successful.

The user should fill all fields of the form, which the application uses for preparing the list “MeasurementsModel” objects. If the form is fully filled and the user presses the “SEND” button, the activity invokes methods of the “SaMiViewModel” that use the central repository to access the database. The repository asynchronously retrieves the stored measurements and sets them in MutableLiveData that is observed by the “SaMiViewModel”. MediatorLiveData of the “SaMiViewModel”, in turn, is observed by the “PostRequestActivity”. In this way the retrieved data is delivered to the activity, where it gets marked with the information from the form. Next, the list with objects of “MeasurementsModel” type is passed to the central repository via the “SaMiViewModel”. The repository uses Retrofit web client to send them to the SaMi server via POST request. If the request is successful, the repository clears the database and notifies the “PostRequestActivity” about that through the “SaMiViewModel”. Afterwards, the activity shows a toast that displays the status of the request and then finishes. Subsequently, the application returns to the “LeConnectedDeviceActivity”.

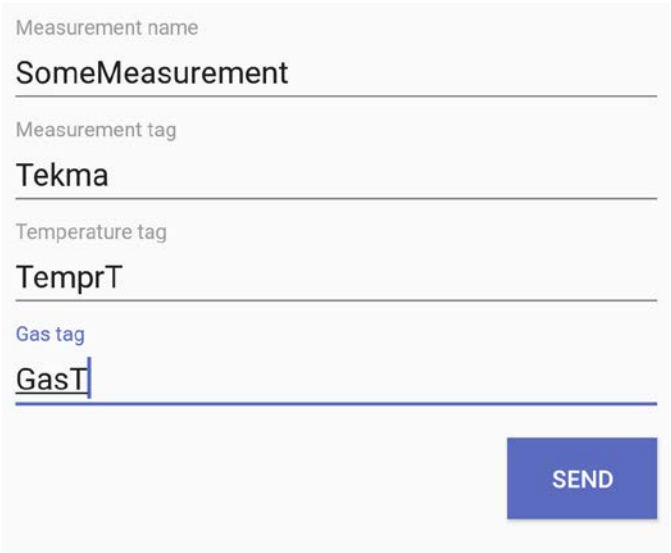

FIGURE 18. Screenshot of the “PostRequestActivity” (self-made)

4.6.9 Activity for GET request and response

Clicking on the “Get data from SaMi cloud” option in the popup menu of the toolbar within “LeConnectedDeviceActivity” starts the “GetRequestActivity”. The activity is divided into two fragments that are aggregated by a ViewPager and a TabLayout, similarly to the “LeConnectedDeviceActivity”.

The first fragment is named “GetRequestBuilder” (Figure 19). It contains a form that allows the user to select the search parameters. To make a GET request, there should be filled at least the ‘key’, the ‘Temperature tag’, and the ‘Gas tag’ fields. As soon as the required fields are complete, clicking the “GET” button passes the fields’ values to the central repository via “SaMiViewModel”. Afterwards, the repository makes the GET request with help of the Retrofit web client. Consequently, the repository is provided with the GET response. If the request status is successful, the GET response contains an object of “MeasurementPackage” type with the measurements retrieved from the server. Next, the actual measurements’ values get withdrawn from this object and then displayed on the graph of the “GetResponse” fragment (Figure 20).

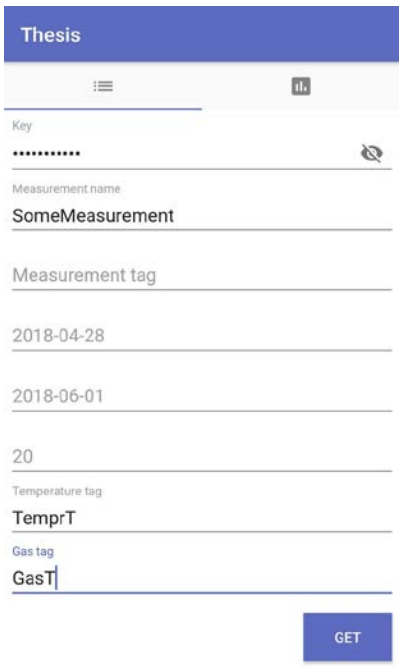

FIGURE 19. Screenshot of
the “GetRequestBuilder” fragment (self-made)

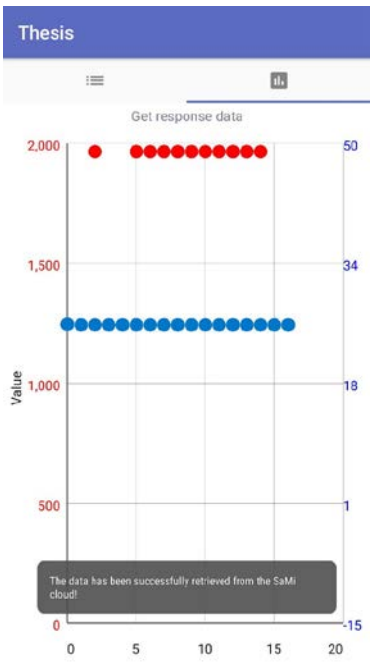

FIGURE 20. Screenshot of
the “GetResponse” fragment (self-made)

5 CONCLUSION

In this work, a fully functioning implementation of the IoT service based on BLE technology has been presented. Firstly, the Arduino Uno specifications and programming of Arduino sketches were studied, followed by the research regarding the BLE shield V2.1. Secondly, the working principles of the temperature sensor V1.2 and the MQ-135 gas sensor were analyzed. Next, the BLE shield was mounted on top of the board, providing it with the capability to exchange the data via BLE. Furthermore, the Arduino libraries, which are required to set up the data acquisition from the sensors and the proper communication line between the board and the shield, were selected. Then the sensors were connected to the board and programmed in accordance with their datasheets. The research concerning the BLE protocol stack was conducted in order to obtain the basic knowledge of the BLE terms, concepts, and working principles. As soon as the board started retrieving the temperature and the carbon dioxide level from the sensors, the BLE-related logic was programmed. Subsequently, the working prototype of a Peripheral BLE device was complete.

The second part of the work comprised the studying of the best practices of the modern mobile software development and the creation of the Android application. Initially, there were the libraries suitable for the needs of the project selected. Afterwards, the MVVM architectural pattern was chosen to build the responsive, robust and efficient application. Then the knowledge regarding the Android BLE API was gained, followed by the establishment of the connection between the application and the Peripheral device. Next, the app was programmed to receive the measurements and output them on the graphs. Moreover, the web client, responsible for the data exchange with the SaMi server was implemented in the application.

As a result, all the goals of the thesis project were successfully achieved. The board is able to retrieve the temperature and the CO_2 values from the sensors and send them to the Android application. The app outputs the measurements on the graph and can exchange the data with the SaMi server. Additionally, the valuable knowledge and experience in the IoT field were acquired.

REFERENCES AND SELF-PRODUCED MATERIALS

BLUETOOTH SIG 2016. Bluetooth Core Specification v5.0 [online publication]. [2018-05-03]. Available:
https://www.bluetooth.com/specifications/bluetooth-core-specification

GEHRING, Jonas 2018. “Summary&Features” tab of the GraphView library website [website]. [Accessed: 2018-05-16]. Available:
http://www.android-graphview.org/

GOOGLE DEVELOPERS. “Accessing data using Room DAOs” guide within the documentation of the Room persistence library [website]. [Accessed: 2018-05-27]. Available:
https://developer.android.com/training/data-storage/room/accessing-data

GOOGLE DEVELOPERS. Activity lifecycle image [digital image]. [Accessed: 2018-05-19]. Available:
https://developer.android.com/guide/components/activities/activity-lifecycle

GOOGLE DEVELOPERS. “Application Fundamentals” guide of the documentation [website]. [Accessed: 2018-05-13]. Available:
https://developer.android.com/guide/components/fundamentals

GOOGLE DEVELOPERS. ConstraintLayout reference in the documentation of the Constraint Layout Library [website]. [Accessed: 2018-05-30]. Available:
https://developer.android.com/reference/android/support/constraint/ConstraintLayout

GOOGLE DEVELOPERS. “Defining data using Room entities” guide within the documentation of the Room persistence library [website]. [Accessed: 2018-05-27]. Available:
https://developer.android.com/training/data-storage/room/defining-data

GOOGLE DEVELOPERS. Fragment lifecycle image [digital image]. [Accessed: 2018-05-20]. Available:
https://developer.android.com/guide/components/fragments

GOOGLE DEVELOPERS. “Fragments” guide of the documentation [website]. [Accessed: 2018-05-20]. Available:
https://developer.android.com/guide/components/fragments

GOOGLE DEVELOPERS. Lifetime of the ViewModel [digital image]. [Accessed: 2018-05-24]. Available:
https://developer.android.com/topic/libraries/architecture/viewmodel

GOOGLE DEVELOPERS. “LiveData Overview” in the description of the “Architecture Components” library [website]. [Accessed: 2018-05-21]. Available: https://developer.android.com/topic/libraries/architecture/livedata

GOOGLE DEVELOPERS. “Projects overview” within the user guide for the Android Studio [website]. [Accessed: 2018-05-13]. Available: https://developer.android.com/studio/projects/

GOOGLE DEVELOPERS. “Save data in a local database using Room” guide within the documentation of the Room persistence library [website]. [Accessed: 2018-05-27]. Available: https://developer.android.com/training/data-storage/room/

GOOGLE DEVELOPERS. “Services overview” guide of the documentation [website]. [Accessed: 2018-05-13]. Available: https://developer.android.com/guide/components/services

GOOGLE DEVELOPERS. “Understand the Activity Lifecycle” guide of the documentation [website]. [Accessed: 2018-05-19]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle

GOOGLE DEVELOPERS. “ViewModel Overview” in the description of the “Architecture Components” library [website]. [Accessed: 2018-05-24]. Available: https://developer.android.com/topic/libraries/architecture/viewmodel

GOOGLE DEVELOPERS. ViewModel reference within the documentation of the “Architecture Components” library [website]. [Accessed: 2018-05-24]. Available: https://developer.android.com/reference/android/arch/lifecycle/ViewModel

IHS Markit, 2017. The Internet of Things: a movement, not a market [online publication]. [Accessed 2018-05-01]. Available: https://cdn.ihs.com/www/pdf/IoT_ebook.pdf

OFFICIAL ARDUINO WEB STORE. Technical specifications of Arduino Uno Rev3 [website]. Arduino Uno appearance [digital photo]. [Accessed 2018-05-02]. Available: https://store.arduino.cc/usa/arduino-uno-rev3

OFFICIAL REDBEARLAB WEB STORE. The appearance of Bluetooth Low Energy shield V2.1 [digital photo]. [Accessed: 2018-05-02]. Available: https://redbear.cc/product/ble-shield.html

REDBEARLAB. ‘README.md’ file of ‘nRF8001’ GitHub repository [website]. [Accessed: 2018-05-15]. Available:
https://github.com/RedBearLab/nRF8001

SAVONIA MEASUREMENTS. Help page of the Savonia Measurements website. [Accessed 2018-05-12]. Available:
https://sami.savonia.fi/Manage/Home/Help

SEEED WIKI. Specifications of the temperature sensor V1.2 [website]. [Accessed 2018-05-10]. Available:
http://wiki.seeedstudio.com/Grove-Temperature_Sensor_V1.2/

SQUARE, INC. 2013. Retrofit guide [website]. [Accessed: 2018-05-17]. Available:
http://square.github.io/retrofit/

VASCONCELOS, Helder 2016. Asynchronous Android Programming Second Edition [book].

WAVESHARE WIKI. The datasheet of MQ-135 gas sensor [website]. [Accessed 2018-05-09]. Available:
https://www.waveshare.com/w/upload/7/71/MQ-135.pdf

WEB STORE OF SEEED TECHNOLOGY CO., LTD. Description of BLE shield V2.1 [website]. [Accessed 2018-05-02]. Available:
https://www.seeedstudio.com/Bluetooth-4.0-Low-Energy-BLE-Shield-v2.1-p-1995.html